Notions avancées sur les listes

Pour parcourir une liste, on peut effectuer un **parcours par indice**. Dans ce cas, la variable de boucle prend successivement tous les indices valides pour cette liste. On utilise `range(len(ma_liste))` pour obtenir les indices de `0` à `len(ma_liste)-1`.

```
ma_liste = ['NSI', 16, 12.4, (1, 4)]
for i in range( len(ma_liste) ) :
     print(ma_liste[i])

# affiche :
# NSI
# 16
# 12.4
# (1, 4)
```

On peut aussi effectuer un **parcours par valeur**. Dans ce cas, la variable de boucle prend successivement toutes les valeurs contenues dans la liste. On n'utilise cette fois pas de range, mais directement la liste.

```
ma_liste = ['NSI', 16, 12.4, (1, 4)]
for e in ma_liste :
     print(e)

# affiche :
# NSI
# 16
# 12.4
# (1, 4)
```

Lorsque l'on passe une liste en paramètre d'une fonction, contrairement aux types de bases, la liste n'est pas copiée. Une **référence** de la liste est passée en paramètre. Toutes modifications faites sur la liste dans la fonction seront persistantes hors de la fonction.

```
ma_liste = [10]
mon_nombre = 1
mon_bool = True

def modif(l, n, b) :
     l.append(30)  # La liste étant passée par référence, les modifications sur
     l[0] = -1     # celle-ci sont persistantes hors de la fonction
     n = 10000     # Les types de bases sont passées par copie, les modifications
     b = False     # dans la fonctions ne sont pas persistantes

modif(ma_liste, mon_nombre, mon_bool)
print(ma_liste, mon_nombre, mon_bool) # affiche [-1, 30], 1, True
```

Exercices

1) Écrire une fonction `recherche_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, et qui parcourt tous les éléments de `l` à la recherche de la valeur `n`.
Si `n` est présent dans `l`, alors la fonction renvoie `True`, sinon elle renvoie `False`.
L'algorithme utilisé par `recherche_occurrence` est donné par le pseudo-code suivant :

```
recherche_occurrence(liste L, entier n) -> booléen:
    On parcourt tous les éléments de L :
        Si l'élément courant est égal à n :
            On retourne Vrai, sinon on continue.
    Le parcours n'a rien trouvé, on retourne Faux.
```

```
print(recherche_occurrence([0, 1, 2, 3, 4], 5)) # affiche False
print(recherche_occurrence([0, 1, 2, 3, 4], 2)) # affiche True
```

2) Écrire une fonction `copie` prenant en paramètre une liste d'entiers `l` et qui retourne une copie à l'identique de la liste `l` passé en paramètre. La liste `l` ne doit pas être modifiée. Il est interdit d'utiliser la méthode `copy()`.

```
l = [1, 2, 3, 4, 5]
l2 = copie(l)
l.pop()
print(l, l2) # affiche [1, 2, 3, 4], [1, 2, 3, 4, 5]
```

3) Écrire une fonction `somme` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la somme de ses éléments.

```
print(somme([-1, -2, -3, -4])) # affiche -10
```

4) Écrire une fonction `produit` prenant en paramètre une liste d'entiers `l`, qui parcourt tous les éléments de `l` pour renvoyer le produit de ses éléments.

```
print(produit([-1, -2, -3, -4])) # affiche 24
```

5) Écrire une fonction `moyenne` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la moyenne de ses éléments.

```
print(moyenne([-1, -2, -3, -4])) # affiche -2,5
```

6) Écrire une fonction `minimum` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la valeur du plus petit de ses éléments.
L'algorithme utilisé par `minimum` est donné par le pseudo-code suivant :

```
minimum(liste L) -> entier:
    valmin := premier élément de L
    On parcourt tous les éléments de L :
        Si l'élément courant est inférieur à valmin :
            valmin := l'élément courant
    On retourne valmin.
```

```
print(minimum([2, 5, 3, 1, 4]))      # affiche 1
print(minimum([-2, -5, -3, -1, -4])) # affiche -5
```

7) Écrire une fonction `maximum` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la valeur du plus grand de ses éléments. La tester.

```
print(maximum([2, 5, 3, 1, 4]))      # affiche 5
print(maximum([-2, -5, -3, -1, -4])) # affiche -1
```

8) Écrire une fonction `empile` prenant en paramètre deux listes d'entiers `l1` et `l2` et qui retourne une nouvelle liste contenant les éléments de `l1` suivis des éléments de `l2`.

```
print(empile([1, 2, 3], [4, 5, 6, 7])) # affiche [1, 2, 3, 4, 5, 6, 7]
```

9) Écrire une fonction `intercale` prenant en paramètre deux listes d'entiers `l1` et `l2` et un entier `i` qui retourne une nouvelle liste contenant les éléments de `l1` dans lesquelles on intercale les éléments de `l2` à partir de l'indice `i`.

```
print(intercale([1, 2, 3], [4, 5, 6, 7], 1)) # affiche [1, 4, 5, 6, 7, 2, 3]
```

10) Écrire une fonction `indice_min` qui prend en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer l'indice du plus petit de ses éléments.

```
indice_min(liste L) -> entier:
    indice := 0
    Pour i de 0 à taille(L) - 1 :
        Si L[i] est plus petit que L[indice] :
            indice := i
    On retourne indice.
```

```
print(indice_min([2, 5, 3, 1, 4]))      # affiche 3
print(indice_min([-2, -5, -3, -1, -4])) # affiche 1
```

11) Écrire une fonction `indice_max` qui prend en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer l'indice du plus grand de ses éléments. La tester.

```
print(indice_max([2, 5, 3, 1, 4]))      # affiche 1
print(indice_max([-2, -5, -3, -1, -4])) # affiche 3
```

12) Écrire une fonction `nombre_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, qui parcourt tous les éléments de `l` et renvoie le nombre d'occurrence de `n`.

```
print(nombre_occurrence([0, 1, 2, 3, 4], 5))    # affiche 0
print(nombre_occurrence([0, 1, 2, 3, 4, 5], 5)) # affiche 1
print(nombre_occurrence([0, 1, 0, 1, 0], 1)) # affiche 2
```

13) Écrire une fonction `indice_premiere_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, qui parcourt tous les éléments de `l` et renvoie l'indice de la première occurrence de `n`.
De la même façon, écrire `indice_derniere_occurence` qui renvoie l'indice de la dernière occurrence.
De la même façon, écrire `indice_occurence` qui renvoie une liste contenant tous les indices des occurrences de `n`.
Testez vos fonctions.

14) Écrivez une fonction `doublon(l)` qui renvoie `True` s'il existe (au moins) un doublon dans `l` et `False` dans le cas contraire.

```
doublon([1, 2, 3, 4]) # affiche False
doublon([3, 2, 3, 4]) # affiche True
```